LECTURE 2
PYTHON PROGRAMMING

SEHH2239 DATA STRUCTURES

LEARNING OBJECTIVES:

- To use Python Functions to structure codes
- To use built-in data structures – Tuple
- Array
- To define classes to represent objects
- To work out tasks using instance methods

FUNCTIONS

FUNCTIONS

- A function is a block of code which only runs when it is called.
- You can pass data, known as parameters, into a function.
- A function can return data as a result.

- Note: pass Statements
  - The pass statement does nothing. It can be used when a statement is required syntactically but the program or function requires no action.

DEFINING A FUNCTION AND CALLING

- Defining a function
  - The keyword def introduces a function definition.
  - It must be followed by the function name and the parenthesized list of formal parameters.
  - The statements that form the body of the function start at the next line, and must be indented.

```
def firstFcn():
  print("Printing from a function.")

def secondFcn(name):
  print("You are " + name)
```

  - Return a value

```
def thirdFcn(n, m):
  p = n+ 2*m
  return p
```

arguments1.py

CALLING A FUNCTION

- To call a function, use the function name followed by parenthesis.

```
firstFcn()

secondFcn("Patrick")
secondFcn("Joseph")
```

- Arguments can be specified after the function name, inside the parentheses. A number of arguments can be added and separate them with commas.
- Calling to get a return value

```
a = thirdFcn(1,2)

print(thirdFcn(2,3))
```

arguments1.py

ARGUMENTS IN FUNCTIONS

- Number of Arguments
  - By default, a function must be called with the correct number of arguments.
- Pass by reference vs value
  - All parameters (arguments) in the Python language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back in the calling function.
- Default arguments
  - A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument.

```
def defaultArgFcn(name, area = "HK"):
  print("You are " + name + " in " + area)

defaultArgFcn("Joseph")
defaultArgFcn("Mary", "Chicago")
```

```
You are Joseph in HK
You are Mary in Chicago
```

TYPES OF ARGUMENTS

- There are two types of arguments.

```
def defaultArgFcn(name, area):
  print("Hello " + name+ ", you live in " +area)
```

Positional arguments

- Positional arguments are values that are passed into a function based on the order in which the parameters were listed during the function definition.

```
defaultArgFcn("John", "Hong Kong")
>> Hello John, you live in Hong Kong
```

Keyword arguments

- Keyword arguments (or named arguments) are values that, when passed into a function, are identifiable by specific parameter names. A **keyword** argument is preceded by a parameter and the assignment operator, = .

```
defaultArgFcn(name ="John", area ="Hong Kong")
>> Hello John, you live in Hong Kong
defaultArgFcn(area ="KLN", name = "Paul")
>> Hello Paul, you live in KLN
```

IMPORT STATEMENT

- You can use any Python source file as a module by executing an import statement in some other Python source file.

```
import module1[, module2[,... moduleN]
```

- For example, Python has a built-in module that you can use for mathematical tasks.

```
import math
print (math.sqrt (9))      3.0
print (math.sqrt (78))     8.831760866327848
print (math.isqrt (68))    8
```

TUPLE

TUPLE

- A tuple is a collection of objects which ordered and immutable.
- Tuples are **sequences**, just like lists.
- The differences between tuples and lists:
  - Tuples **cannot be changed** (immutable) unlike lists
  - Tuples use parentheses (), whereas lists use square brackets [].
- Creating a tuple is as simple as putting different comma-separated values.

```
t1 = ("apple", "box", "cake", "disk")
print(t1)
>>('apple', 'box', 'cake', 'disk')
t2 = (1, 2, 3, 4, 5 )
t3 = "a", "b", 6, 0x23
print(t3)
>>('a', 'b', 6, 35)
t4 = ()
```

tuple.py

ACCESS ITEMS AND REMOVING TUPLES

- Access items
  - Tuple items by referring to the index number, inside square brackets:

```
print(t1[2])
>> cake
print(t2[1:3])
>>(2, 3)
```

- Updating Tuples or deleting an item
  - Tuples are immutable which means you cannot update or change the values of tuple elements.
    - Removing individual tuple elements is not possible.
- Remove an entire tuple

```
del t2
```

BASIC TUPLES OPERATIONS

```
t1 = ("apple", "box", "cake", "disk")
```

Python Expression	Results	Description
len(t1)	4	Length
t1+("egg", "fish")	('apple', 'box', 'cake', 'disk', 'egg', 'fish')	Concatenation
("go!",) * 3	('go!', 'go!', 'go!')	Repetition
"box" in t1 "ox" in t1	True False	Membership
for x in (2, 3, 6): print(x)	2 3 6	Iteration

INDEXING, SLICING, MATRIXES, MIN AND MAX

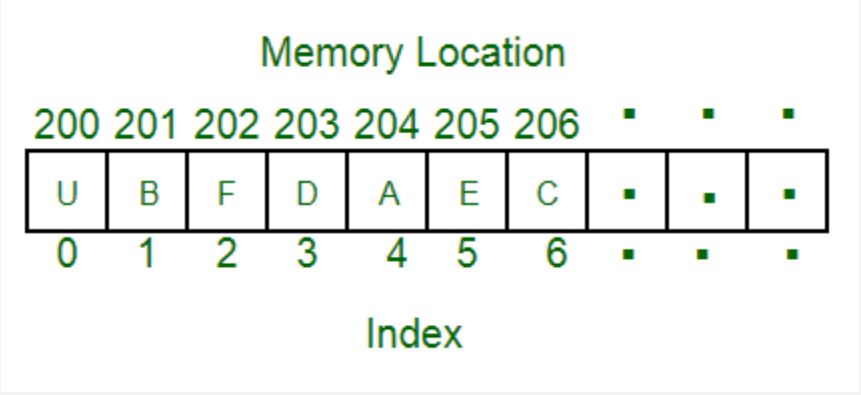
```
t1 = ("apple", "box", "cake", "disk")
t2 = (1, 2, 3, 4, 5 )
```

Python Expression	Results	Description
t1[3]	'disk'	Index start at zero
t1[-2]	'cake'	Negative: count from the right
t1[2:]	('cake', 'disk')	Slicing fetches sections
t1[::2]	('apple', 'cake')	Slicing
t1[::-1]	('disk', 'cake', 'box', 'apple')	Listing from the back
max(t2)	5	Max
min(t2)	1	Min

ARRAYS

ARRAY DEFINITION

- An ***array*** is a sequenced collection of variables all of the same type.
- Each variable, or ***cell***, in an array has an ***index***, which uniquely refers to the value stored in that cell. The cells of an array, Arr1, are numbered 0, 1, 2, and so on.
- Each value stored in an array is often called an ***element*** of that array.



- Array is the simplest data structure where each data element can be randomly accessed by using its index number.

ARRAYS IN PYTHON

- Python does not have built-in support for Arrays, but **Lists** can be used instead.
- To declare a variable that holds an array of strings as a **list**.

- To insert values to it, we can use an array literal - place the values in a comma-separated list, inside square brackets:

```
sports = ["soccer", "basketball", "badminton", "squash"]
```

- To create an array of integers, you could write:

```
n = [ 10, 20, 35, 40]
```

ACCESS THE ELEMENTS OF A LIST

- You access a list element by referring to the index number.
- This statement accesses the value of the first element in sports:

```
print(sports[2])
```

```
>>badminton
```

CHANGE AN ELEMENT IN A LIST

- To change the value of a specific element, refer to the index number:
- Example

```
sports[2] = "tennis"
print(sports[2])
```

```
>> tennis
```

- Use negative index to access elements from the end.

```
sports[-1]
```

```
>>squash
```

```
sports[-3]
```

```
>>basketball
```

LENGTH OF A LIST

- To find out how many elements a list has, use the *len* property:

- Example:

```
len(sports)
```

```
>> 4
```

LIST SIZE

- Generally, a list is of fixed size and each element is accessed using the indices.
  - For example, we have created a list with size 4.
  - Then the valid expressions to access the elements of this list will be `sports[0]` to `sports[3]` (length-1).
  - Whenever you used the value greater than or equal to the size of the list (e.g. `sports[9]` or `sports[20]`), it gives

    `IndexError: list index out of range.`

- Caution: The following is not correct.

```
sports[6] ## It will give an error
>>>>>> IndexError: list index out of range
```

LOOP THROUGH A LIST

- To show the all content in a list, just print the variable name.

```
print(sports)
```

```
>> ['soccer', 'basketball', 'badminton', 'squash']
```

- The following example outputs all elements in the **sports** list with for loop:
- The length property to specify how many times the loop should run.

- Example

```
for a in sports:
  print(a)
```

```
>>
soccer
basketball
tennis
Squash
```

LIST OPERATIONS

- Add a new element

```
sports.append("swimming")
sports
```

```
>>['soccer', 'basketball', 'badminton', 'squash', 'swimming']
```

- Return and Remove an element at the specified position

```
#Remove an element
sports.pop(2)
```

```
>>badminton
```

```
sports
```

```
>>['soccer', 'basketball', 'squash', 'swimming']
```

LIST OPERATIONS

Remove the first occurrence of the element with the specified value

```
sports.remove("squash")
sports
```

```
>>['soccer', 'basketball', 'swimming’]
```

Return the position at the first occurrence of the specified value.

```
sports.index("swimming")
```

```
>> 2
```

Inserts the specified value at the specified position.

```
sports. insert(2, "hockey")
sports
```

```
>>['soccer', 'basketball', 'hockey', 'swimming']
```

LIST COPYING

Two ways to copy lists :

1. Simply using the assignment operator.
2. Deep Copy

1. Simply using the assignment operator.

Assignment statements do not copy objects, they create bindings between a target and an object.

```
sports = ["soccer", "basketball", "badminton", "squash"]
new_sports = sports
print(id(sports))
>> 140211906678128
print(id(new_sports))
>> 140211906678128
sports[1] = "polo"
print(new_sports[1])
>>polo
```

LIST COPYING

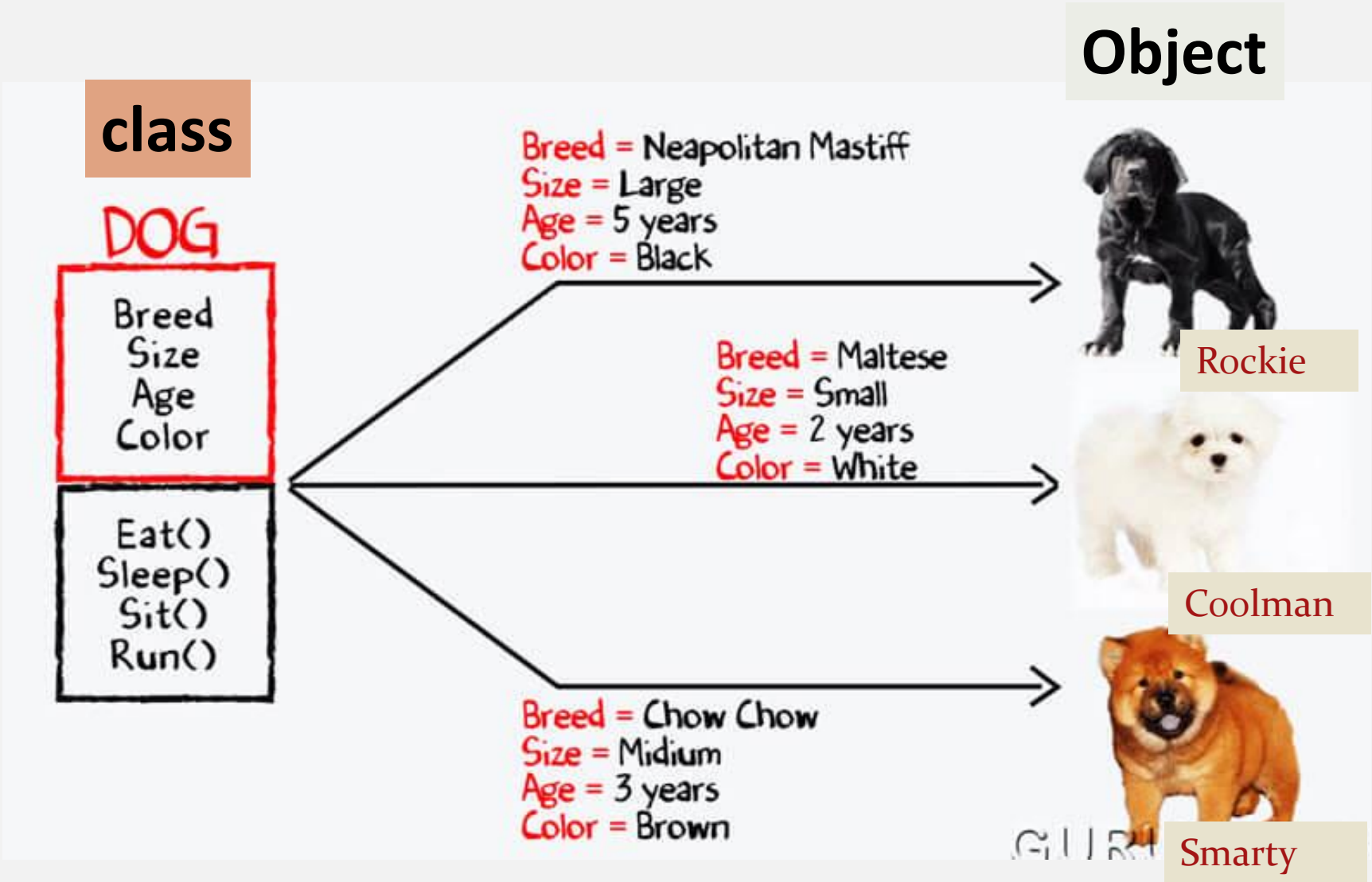
2. Deep Copy

A copy of the object is copied into another object. It means that any changes made to a copy of the object do not reflect in the original object.

```
sports = ["soccer", "basketball", "badminton", "squash"]
new_sports = sports.copy()
print(id(sports))
>> 140211906678128
print(id(new_sports))
>> 140211801453104
sports[1] = "polo"
print(new_sports[1])
>>basketball
```

FUNDAMENTALS OF OBJECT-ORIENTED PROGRAMMING IN PYTHON

CLASS AND OBJECTS



CLASS DEFINITIONS

- A **class** serves as the primary means for abstraction in object-oriented programming.
- A **class** provides a set of *behaviors* in the form of member functions (also known as **methods**), with implementations that belong to all its instances.
- A **class** also serves as a blueprint for its instances, effectively determining the way that state information for each instance is represented in the form of **attributes** (also known as **fields**, **instance variables**, **object variables** or **data members**).

CLASS AND OBJECTS

- Each **object** created in a program is an **instance** of a **class**.
- Each class presents to the outside world a concise and consistent view of the objects that are instances of this class, without going into too much unnecessary detail or giving others access to the inner workings of the objects.
- The class definition typically specifies **instance variables**, also known as **data member**s, that the object contains, as well as the **methods**, also known as **member functions**, that the object can execute.

DEFINE A CLASS IN PYTHON

- All class definitions start with the `class` keyword, which is followed by the name of the class and a colon (:).
  - By conventionPython class names are written in CapitalizedWords notation, such as Cinema, WineGlass, OfficeForIT.
- The properties that all `Dog` objects must have are defined in a method called `__init__()`.
  - Every time a new Dog object is created, `.__init__()` sets the initial **state** of the object by assigning the values of the object's properties. That is, `__init__()` initializes each new instance of the class.
  - You can give `__init__()` any number of parameters, but the first parameter will always be a variable called `self`.
  - When a new class instance is created, the instance is automatically passed to the `self` parameter in `__init__()` so that new attributes can be defined on the object.

```
class Dog:

    def __init__(self, name, age):

        self.name = name

        self.age = age
```

oop1.py

INSTANTIATE AN OBJECT IN PYTHON

- Creating a new object from a class is called *instantiating* an object.
- You can instantiate a new Dog object by typing the name of the class, followed by opening and closing parentheses ().

```
d1 = Dog("Rockie", 5)
d2 = Dog("Coolman", 2)
```

- This creates two new Dog instances.
- When you instantiate a Dog object, Python creates a new instance and passes it to the first parameter of `__init__()`.
- This essentially removes the `self` parameter, so you only need to worry about the name and age parameters.

ACCESS INSTANCE ATTRIBUTES

- Instance attributes are accessed using **dot notation**:.

```
d1.name
d1.age

d2.name
d2.age
```

```
Rockie
5

Coolman
2
```

INSTANCE METHODS

- Instance methods are functions that are *defined inside a class* and can only be called from an instance of that class.
- This Dog class has two instance methods:
  - .description()

    returns a string displaying the name and age of the dog.
  - .eat()

    has one parameter called food and returns a string containing the dog's name and the food the dog eats.

```
class Dog:

    def __init__(self, name, age):
        self.name = name
        self.age = age

    # Instance method
    def description(self):
        return f"{self.name} is {self.age} years old"

    # Another instance method
    def eat(self, food):
        return f"{self.name} eats {food}"
```

```
d1 = Dog("Rockie", 5)
```

```
d1.description()
```

```
'Rockie is 5 years old'
```

```
d1.eat("pork")
```

```
'Rockie eats pork'
```

```
d1.eat("snacks")
```

```
'Rockie eats snacks'
```

oop2.py